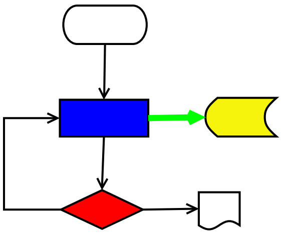
VICI



Visual Chart Interpreter

Writing Tests

Publication History

Date	Who	What Changes
18 May 2018	Brenton Ross	Initial version.

Table of Contents

1 Introduction....5
- 1.1 Scope....5
- 1.2 Aim....5
- 1.3 Overview....5
- 1.4 Audience....5

2 The Test Manager....6
- 2.1 Configuring....6
- 2.2 Installing and Running Tests....6
- 2.3 Creating the Test Report....7

3 Test, Scenarios and Test Cases....8
- 3.1 Test Class....8
- 3.2 Scenario Class....8
- 3.3 Test Case Class....8
- 3.4 A Test Method....9

4 Asynchronous Test Cases....10
- 4.1 Overview of Events....10
- 4.2 Instrumenting the Code....10

5 Plug-In Tests....11
- 5.1 Creating an Autorun Plug-In....11
- 5.2 Building a Test Plug-In....12
- 5.3 Loading and Running Plug-Ins....12
- 5.4 Discovering Program Objects....14
  - 5.4.1 Making Objects Discoverable....14
  - 5.4.2 Finding Discoverable Objects....14

6 Testing GUI Programs....15
- 6.1 Preparing a Program for GUI Testing....15
- 6.2 Creating a Test Case....16
- 6.3 Creating a Test Script....16
- 6.4 Running a GUI Test....16

7 Using The gth-editor Program....17
- 7.1 Configuration....17
- 7.2 Selecting a Test....17
- 7.3 Allocating Tests to Windows....18
- 7.4 Defining Actions for a Test Case....19
- 7.5 Writing Lua Scripts....21
- 7.6 Testing the Script....22
- 7.7 Testing Graphics Scenes....22
- 7.8 Adaptor Reference....22

8 Interface Testing....25
- 8.1 Installing Test Modules....25
- 8.2 Simulating Events....25
- 8.3 Event Tracing....26

9 Testing Installed Programs....27

10 Logging....28
10.1 Network Logging....28
11 Tracing....29
11.1 Trace Macro....29
11.2 Trace Logging....29
11.3 Trace to Dot....29
Appendix A....30

1 Introduction

This document is a guide to writing tests for VICI. It should be read in conjunction with the VICI Test Plan document.

1.1 Scope

The document covers writing test programs with test cases. The use of plug-in libraries to hold the tests is described.

The writing of tests for GUI programs using the gth-editor will be explained.

There is a section on writing tests to ensure the interfaces between the modules of VICI are working correctly.

Testing of the programs for various combinations of libraries, and also for various Linux distributions and desktops is also covered.

The document will also include guidelines that the programs should conform to so as to make testing easier.

Finally, as an aid to debugging the use of the trace facility is described.

1.2 Aim

The aim of writing tests is to first confirm that the new code conforms to the design specification and subsequently to confirm that changes to the code base and changes to the 3rd party libraries do not cause the programs to break.

1.3 Overview

The first section describes what needs to be done to use the test manager. This is responsible for running the tests and producing a report of the results.

Basic test cases are described in the next section.

Asynchronous test cases are described in section 4.

How to set up plug-in tests is covered in section 5

Section 6 covers setting up GUI tests, while the subsequent section provides a reference for the gth-editor.

Interface tests for testing the modules in isolation is covered next.

The setting up and running of a Test Lab for testing multiple distributions is shown in section 9.

The last sections shows how to use the logging and tracing modules.

1.4 Audience

This document is for the programmers and testers of VICI.

2 The Test Manager

The test manager is responsible for running the tests and producing a report. A test is made up of a Test object, one or more Scenario objects, each of which has one or more TestCase objects. Each of these uses its constructor to set up the test and its destructor to clean up. Each of these also has an associated factory class that is held by the test manager and used to create the test object when it's needed.

2.1 Configuring

The test manager is implemented in the Tester class. This uses the singleton pattern.

Once the main XINI configuration has been set up, the configure method is called passing the name of the test.

```
cdi::Tester &tester = cdi::Tester::instance();
tester.configure("libcfi");
```

The Tester gets its configuration from the XML configuration file:

```
<Test>
      <libcfi>
             <Report>
                    <name>TestLog</name>
             </Report>
      </libcfi>
</Test>
```

2.2 Installing and Running Tests

The scenarios and test cases are installed using a static method defined in each. (See below)

Once they are installed the tests can be run. They are run in alphabetical order of the scenario and test case names which are just arbitrary strings. (Not applicable for GUI tests.)

```
tester.runTests();
```

2.3 Creating the Test Report

The “name” in the configuration for the test refers to an entry in the configuration for logging.

```
<LOG>
      <TestLog>
            <type>unformatted</type>
            <file>@abs_top_builddir@/scratch/logs/test.log</file>
      </TestLog>
</LOG>
```

This example is drawn from an XML configuration that is generated for doing the local make check.

The test report is created by calling the tester's summary method.

```
bool rc = tester.summary();
```

3 Test, Scenarios and Test Cases

3.1 Test Class

If you need to create a Test object, then derive it from cdi::TestT<>. It should be installed using its static install() method.

If you do not specify a Test object a default one will be installed.

3.2 Scenario Class

The scenario class is illustrated in the following example.

```
class EditScenario : public cdi::ScenarioT<EditScenario>
{
private:
      Ed::ViciEditor *ge;
public:
      EditScenario(csr nm);
      ~EditScenario();
};
```

It uses the “Curiously Recurring Template Pattern” wherein the class is derived from a template class that has the class as it parameter. This pattern is used to automatically create a static install() method that applies to each derived class.

The scenario object can get a pointer to the Test object if it should need access to the data set up by the test. A dynamic cast will be required.

The scenario classes are installed using their static install() method which takes a string parameter that is the name of the scenario. Scenarios are run in alphabetical order.

If the test cases install themselves with a reference to a scenario that has not been installed then a default scenario will be installed.

3.3 Test Case Class

The basic test case class is illustrated in the following example:

```
class XmlTestCase : public cdi::TestCaseT<XmlTestCase>
{
protected:
      bool runTest();
public:
      XmlTestCase(csr nm) : cdi::TestCaseT<XmlTestCase>(nm) {}
      ~XmlTestCase(){}
};
```

It uses the “Curiously Recurring Template Pattern” wherein the class is derived from a template class that has the class as it parameter. This pattern is used to automatically create a static install() method that applies to each derived class.

The illustrated test case doesn't do any setting up or cleaning up so the constructor and destructor are empty. In a more complex case you would use them to prepare for the tests.

The test cases are installed using their install() method which takes two string parameters – the name of the test case, and the name of the scenario it should run under. They are run in alphabetical order.

```
ParseTreeTestCase::install("parseTree", "Parse Tree");
```

The test cases can get pointers to the test object and the current scenario object by calling methods on Tester. A dynamic cast will be required.

The runTest() method contains the code to perform the tests. It should return false if the test failed.

3.4 A Test Method

A typical test method in a test case is illustrated in the following example

```
bool MyTestCase::runTests()
{
      bool rc = true;
      rc = rc && test( a == b, “equals test” );
      rc = test( a!=c, “unequal test” ) && rc;
      return rc;
}
```

The test() method returns the value of its first (boolean) parameter. It also records the test result in the scenario object so that a report can be produced. The report will include the second parameter if the test should fail, hence you should ensure they are unique so you can locate failed tests.

Occasionally you may need to run a test which is expected to fail. Adding a third parameter EXPECTED will prevent the failure from terminating the testing.

By combining the test result with the current return code you can control whether or not to run tests if previous tests have failed.

4 Asynchronous Test Cases

These are used to test the state of the system when a callback method has been activated. Typically this will be after calling a function which triggers activity in a separate thread or a child process. It could also be used when waiting for a human response.

The AsynchTestCase is derived from TestCase and should be installed in the same way. Its runTests() method has been overridden to provide three methods that need to be implemented.

4.1 Overview of Events

The test case sets asyncTestEvent to point to the TestEventQueue::event() method. (It points to an empty method, otherwise.)

The test case's initTest() method is called first. This triggers the action in the other thread or process, which, in turn, causes a callback method to be run.

The callback calls the asyncTestEvent() method. This puts an event on a queue and then waits for a notification.

The test case's handleEvent() method is called. This should perform the required tests. The last expected event should set the variable “done” to true.

The test case then notifies the callback's asyncTestEvent() method and processing resumes.

A timeout() method is called if the callback is not called within 10 seconds.

The asyncTestEvent function pointer is returned to its former state, normally to an empty function which exists in log.cpp.

4.2 Instrumenting the Code

Supporting asynchronous testing requires some small changes to the code being tested.

The callback method must call the asyncTestEvent method. This is declared in test.h.

```
(*asyncTestEvent)("first");
```

The parameter can be used to inform the handleEvent method about the sequence of the callback and may also include encoded values from local variables.

The handleEvent() method is passed a pointer to a cdi::TestEvent object. The string passed to asyncTestEvent can be accessed via the event's id() method.

5 Plug-In Tests

Plug-In tests are used to test programs in their completed state. The preceding tests are usually performed by test programs that load libraries containing the code to be tested. When there are no libraries, or when we want to test a program using several of these libraries the plug-in tests can be used.

The cfi library includes support for plug-in libraries, and for testing we use the autorun type which gives the program a form of “power on self test”.

Setting up the test cases is unchanged for plug-ins.

5.1 Creating an Autorun Plug-In

The plug-in class is derived from cfi::AutoRunPlugIn

```
class AdminTestPlugin : public cfi::AutoRunPlugIn
{
public:
      virtual void execute();
};
```

The execute() method is where we set up and run the tests using the classes and methods described in the preceding sections.

The plug-in requires a few global objects:

```
std::list< PlugInFactory * > factories;
```

```
void initViciPlugin()
{
      PlugInDetails details;
      details.pluginFamily = "admin test harness";
      details.name = "admintest";
      details.version = VICI_PLUGIN_VERSION;
      details.description = "Test plugin for testing the vici-admin program.";
      details.factory = new PlugInFactoryT< AutoRunPlugIn, AdminTestPlugin>;

      factories.push_back( details.factory );
      PlugInMgr::instance().regn(details);
}
```

```
void closeViciPlugin()
{
      for ( auto &P : factories )
      {
            delete P;
      }
}
```

5.2 Building a Test Plug-In

Autotools do not support building libraries that are only for use in testing, so they have to be built along with the main libraries. They should be installed in a “plugin” subdirectory under libdir. (They don't need to be on the library path as they will be found via the configuration file.)

The Makefile.am declaration of the plugin library:

```
pkgpluginlibdir=$(pkglibdir)/plugin
pkgpluginlib_LTLIBRARIES=libadmintest.la
```

The plug-in will need to pull in the test libraries as they should not be part of the program under test.

```
libadmintest_la_LIBADD = -lcdi -lgth
```

The remaining entries for the plug-in are the usual ones for any library.

5.3 Loading and Running Plug-Ins

For non-GUI programs it is just necessary to have the plug-in manager load the test plug-in using an entry in the configuration file:

```
PlugInMgr::instance().load("plugintest");
```

This can be called as soon as the configuration file has been loaded.

```
<PlugIns>
      <PlugIn prog='plugintest'>
            <name>plugtest</name>
            <mode>autorun</mode>
            <path>@abs_top_builddir@/UnitTest/.libs/libplugtest.so</path>
      </PlugIn>
</PlugIns>
```

The prog attribute is the name passed to the load() method of the PlugInMgr while the name entity contains the name of the plug-in within the library. (There can be more than one plug-in in a library.)

The path used here is for a plug-in that has not been installed. This is necessary for doing make check.

For GUI programs we need to delay the running of the tests until the widgets have been realised.

The class VMainWindow is derived from the Qt QMainWindow and has a startUp() method which is called after show() has been run.

```
void MainWindow::startUp()
{
      try {
             // check for plug-ins
             cfi::PlugInMgr::instance().load("vici-admin");
      }
      catch( vx & xx)
      {
             cerr << xx.what() << endl;
      }
}
```

5.4 Discovering Program Objects

One of the problems with the plug-in approach is how to discover the objects that we want to test. The program objects cannot rely on the test objects being loaded, and the test objects have no links to the program objects.

5.4.1 Making Objects Discoverable

Header files that declare objects that need to be discoverable must include “discover.h”.

Each discoverable class should be privately derived from cfi::Discoverable as shown in this example:

```
class ViciAdminImpl : public VICI::Admin::ViciAdmin,
                      public AdminInterface,
                      private cfi::Discoverable
{
…
};
```

The constructors for these classes must include the macro DISCOVERABLE near their start to register their existence with the DiscoveryMgr object.

5.4.2 Finding Discoverable Objects

The test code will need to include dicover.h

A test case can then locate a discoverable object using code similar to this example:

```
vector< void * > objs;
DiscoveryMgr::instance().fetch("VICI::Admin::ViciAdminImpl", objs);
if ( objs.size() > 0)
{
      admin = static_cast<Admin::ViciAdminImpl *>(objs[0]);
}
```

6 Testing GUI Programs

To test a GUI program we need to simulate the pressing of keys and the actions of the mouse to manoeuvre the program into various states where its internal values can be examined. The GUI Test Harness provides this capability.

The core object is GTHTest which is derived from Scenario but has been extended to tke over running the test cases from Tester. This is necessary since the order of running the test cases needs to be controlled.

The constructors and destructors of each test case are used to provide a sequence of actions that are applied to the GUI. The body of each test case is written as normal.

6.1 Preparing a Program for GUI Testing

Some minor changes are required of the program to be tested to support GUI testing.

The programs must include gth.h.

The MainWindow class should be derived from VMainWindow, declared in libgui.h. This pulls in the GTHWindowWidget class which declares a function used to register widgets:

```
virtual void RegnFn( std::function<void (void *, gth::WidgetType, csr )> reg )
```

Note that the main window should include MainWindow in its class name so that the test harness can tell when its actually got the main window.

The RegnFn must be implemented and should register the widgets:

```
void MainWindow::RegnFn( std::function<void (void *,
                          VICI::gth::WidgetType, csr )> reg )
{
      reg(exitAct,   gth::Action, "ExitAct");
      reg(saveAct,   gth::Action, "SaveAct");
      reg(helpAct,   gth::Action, "HelpAct");
      reg(qtHelpAct, gth::Action, "QtHelpAct");
}
```

Similarly dialogs should be derived from VDialog and include a registration function.

Declared in libgui is a WidgetMgr class (a singleton) that the windows register themselves with. **For this to work correctly each window and dialog must have a unique window title.**

The WidgetMgr provides a pointer to each window or dialog to the test harness when they first show. The test harness then calls the registration function to get pointers to the widgets.

6.2 Creating a Test Case

The following example illustrates how to create a test case:

```
class TestCaseOne : public gth::GTHTestCase,
                          public cdi::TestCaseT<TestCaseOne>
{
protected:
      virtual bool runTest();
public:
      TestCaseOne(csr);
      ~TestCaseOne();
};
```

Note that GTHTestCase should be first.

The constructor and destructor of GTHTestCase interact with GTHTest to set up the GUI. Additional set up can be done in the class's constructor but is rarely necessary.

6.3 Creating a Test Script

The test harness gets the path name for the test script from the configuration file. If the file does not exist it will be created. If it does exist it may be updated if new windows appear during the test.

```
<Test>
      <libgth>
            <script>${DOLLAR}VICI/src/gth/TestProgram/test/gthtest.xml</script>
            <Report><name>TestLog</name></Report>
      </libgth>
</Test>
```

(The ${DOLLAR} construct is used to get the envsubst program to insert a $ prefix for the VICI environment variable.)

The script should be updated using the gth-editor.

6.4 Running a GUI Test

It should be possible to run the GUI tests as a target in make check, from the Eclipse menus, from a terminal, or from the Run option in the gth-editor.

While a test is running it will grab the mouse which makes other tasks problematic. The F5 function key can be used to pause and resume a test.

7 Using The gth-editor Program

7.1 Configuration

The XML configuration file for the gth-editor used in development is a bit special since it needs to refer to all the VICI sub-projects. This sort of breaks the independence of the sub-projects but is only a configuration file.

TODO Add a para on the configuration for testing the installed programs.

```
<Test>
      <admintest>
             <config>$VICI/src/admin/build/admin.xml</config>
             <script>$VICI/src/admin/TestProgram/test/admintest.xml</script>
             <Report><name>TestLog</name></Report>
      </admintest>
      <libgth>
             <config>$VICI/src/gth/build/gth.xml</config>
             <script>$VICI/src/gth/TestProgram/test/gthtest.xml</script>
             <Report><name>TestLog</name></Report>
      </libgth>
      <search-ui>
             <config>$VICI/src/search/build/search.xml</config>
             <script>$VICI/src/search/TestProgram/test/searchtest.xml</script>
             <Report><name>TestLog</name></Report>
      </search-ui>
      <syntax-gth>
             <config>$VICI/src/syntax/build/syntax.xml</config>
             <script>$VICI/src/syntax/TestProgram/test/syntaxtest.xml</script>
             <Report><name>TestLog</name></Report>
      </syntax-gth>
</Test>
```

In addition to the script and Report entries there is a config entry which points to the configuration file that should be passed to the test program when it is run from within gth-editor.

7.2 Selecting a Test

Use Open to select a test. It will display a selection list for the entries in the configuration.

Confirm the correct test has been opened by checking the details in the resultant message box.

7.3 Allocating Tests to Windows

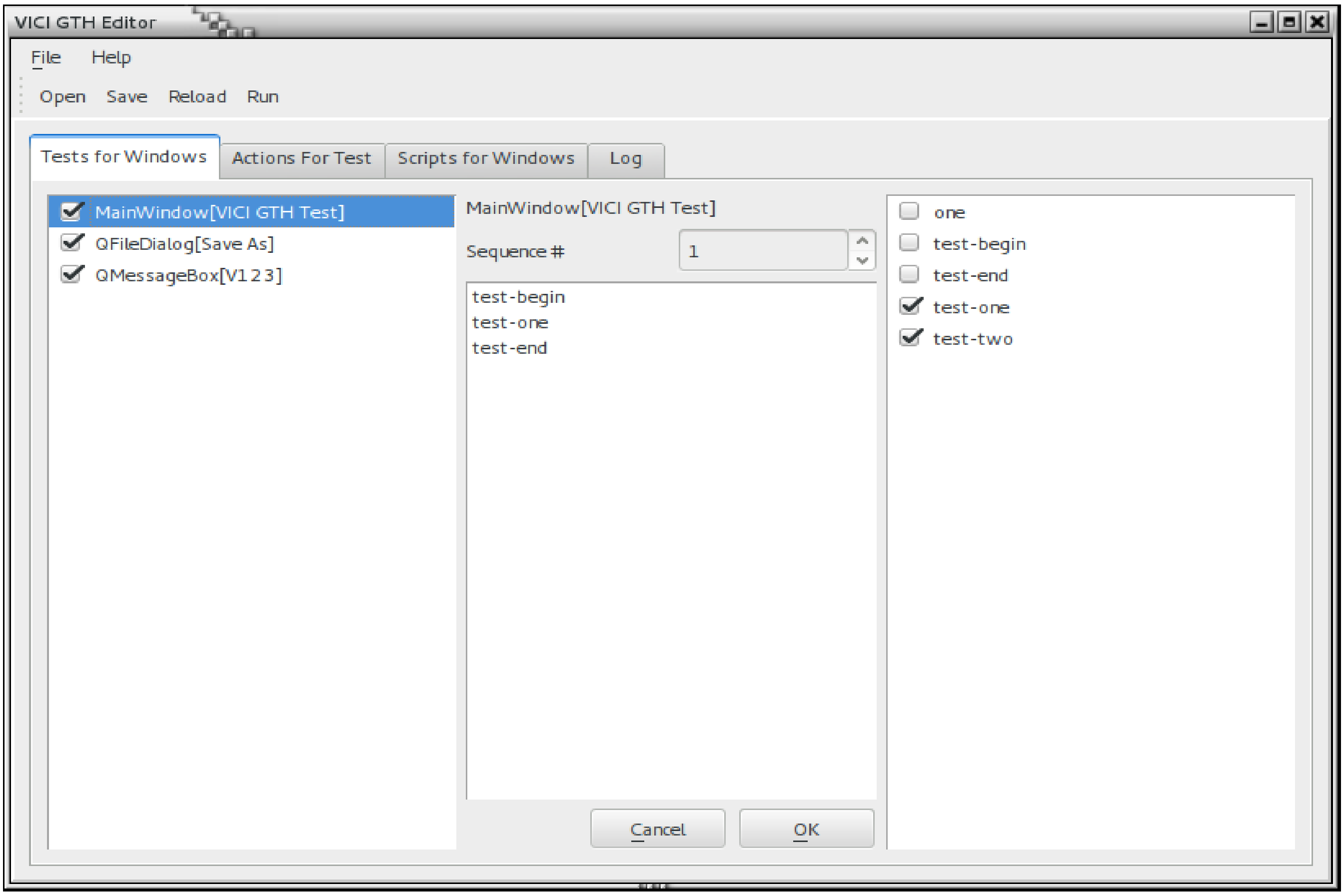
Selecting the “Test for Window” tab brings up the view shown below.

On the left is a list of the windows that are known to the script. They will be checked if they are referenced in the script. If the script should show a window for which no tests are defined it will add it to list as an unchecked entry.

On the right is the list of test cases. Those that are checked correspond to ones that are defined in the test harness code. Unchecked ones are “virtual” test cases that are automatically created just to run actions on the GUI.

The centre column lists the tests allocated to the selected window. Double clicking an entry on the right will add it to the end of the list.

What is the Sequence # ?



☑ MainWindow[VICI GTH Test]
☑ QFileDialog[Save As]
☑ QMessageBox[V1 2 3]

MainWindow[VICI GTH Test]

Sequence # 1

test-begin
test-one
test-end

□ one
□ test-begin
□ test-end
☑ test-one
☑ test-two

7.4 Defining Actions for a Test Case

Selecting the “Actions for Test” tab brings up the view shown below.

The left column selects the test case (with virtual ones unchecked) and at the top of the main panel you can select Constructor or Destructor to specify when the actions should be executed.

The list shows the actions defined for the test case's constructor or destructor. Selecting a row from the list displays it in the fields below where they can be edited.

Label is used to specify the target for a jump. Only forward jumps are allowed.

Delay specifies the time in milliseconds before running the action.

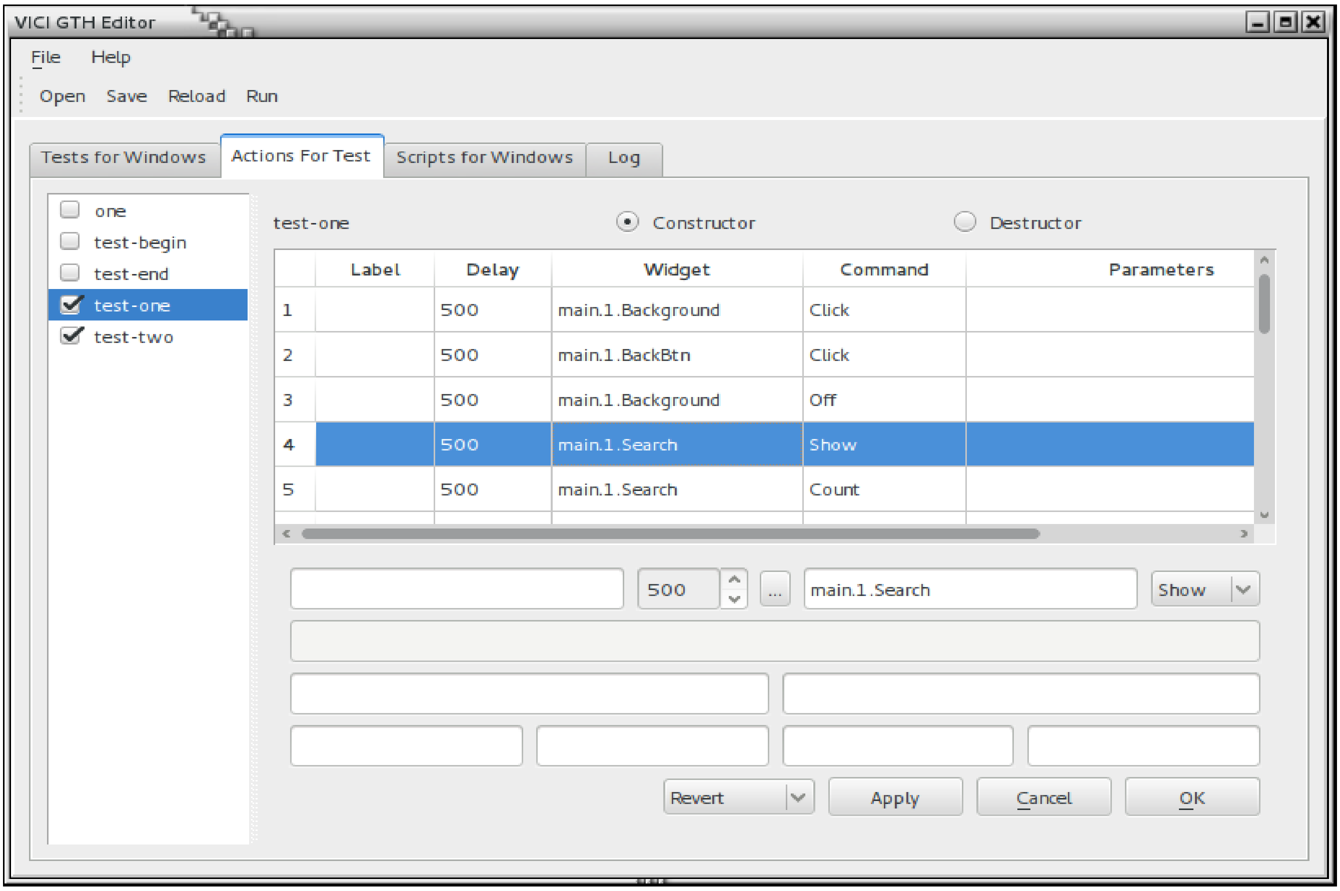
Widget is the name assigned to the widget by the registration function.

Command is the name of the action to perform. These are dependent on the type of the widget. Parameters shows those assigned to the command.

The ellipses button brings up a selection list for the widgets.

The combo box shows the actions available for the widget based on its type as specified in the registration function.

The next row has up to 4 entry fields for parameters. This will depend on the action.



☐ one
☐ test-begin
☐ test-end
☑ test-one
☑ test-two

test-one ● Constructor ○ Destructor

	Label	Delay	Widget	Command	Parameters
1		500	main.1.Background	Click	
2		500	main.1.BackBtn	Click	
3		500	main.1.Background	Off	
4		500	main.1.Search	Show	
5		500	main.1.Search	Count	

500 main.1.Search Show

On the next row the left field is for a regular expression that will be applied to the results of the action. The right field contains a comma separate list of variable names that are assigned the values from the regular expression.

These variables are made available to the test case via the getVal() method of GTHTestCase. They can also be referenced from the Lua scripts.

The bottom row of entry fields are two pairs each containing a regular expression and a label. If the expression should match then the processing skips to the corresponding label.

The following can be a bit confusing to use

The drop down at the bottom selects what is to be done to the edited values:

Revert – abandons the changes

Append – adds to the end of the list

Update – replaces the selected action

Insert Before – inserts as a new action before the selected action

Insert After – inserts as a new action after the selected action

Delete – removes the selected action

Move Up – moves the selected action up one row

Move Down – moves the selected action down a row.

The Apply button applies the action.

The Cancel button undoes any changes and reverts to the previous (?) state.

The OK button saves the updated actions to the programs internal buffers.

(Use the Save button to write the test script to disk.)

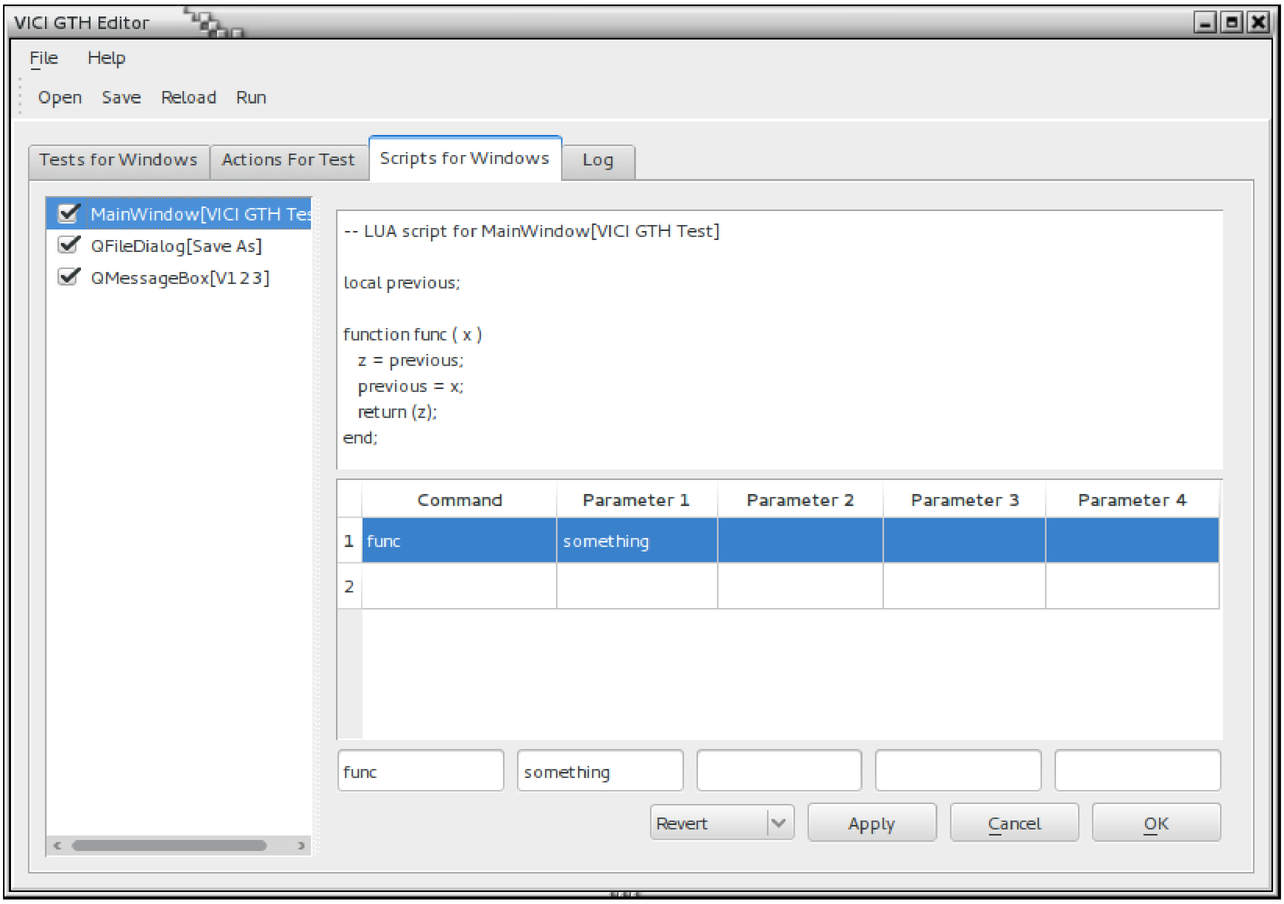
7.5 Writing Lua Scripts

When the logic gets too complex for simple jumps you can add a Lua script. The “Scripts for Windows” brings up the tab for entering and editing the scripts.

A built-in function “Version” can be used to determine the version of Lua. This allows you to modify the script accordingly. It will return zero if Lua is not available.

The chunk of script is associated with a window. This chunk may have any number of functions, and each function may have up to 4 string parameters.

Below the code entry window is a list and a set of fields for editing the entries in the list. Add the function names and their parameter names to the list so that they can be selected when defining actions for a test case.



☑ MainWindow[VICI GTH Tes
☑ QFileDialog[Save As]
☑ QMessageBox[V1 2 3]

```
-- LUA script for MainWindow[VICI GTH Test]

local previous;

function func ( x )
   z = previous;
   previous = x;
   return (z);
end;
```

	Command	Parameter 1	Parameter 2	Parameter 3	Parameter 4
1	func	something			
2					

func | something

7.6 Testing the Script

After editing the script select “Save” to write the script back to the disk. You should have the script under version control.

Select “Run” to test the script. The “Log” tab will be displayed and will show the logging output for the script.

Once the script has completed, select “Reload”. This is important since the test may have updated the script with new details, such as additional windows or dialog boxes.

7.7 Testing Graphics Scenes

It is a bit limited at the moment. You can move to or click at some point in scene coordinates.

A Mouse command turns on an event filter that reports the position of a mouse click in scene coordinates to help in setting up the test.

7.8 Adaptor Reference

Widget Type	**Command**	**Parameters**	**Result**
Action	Trigger		OK
Button	Click		OK
	Get		Label
CheckBox	Click		OK
	On		OK
	Off		OK
	Get		Checked, Unchecked, Partially Checked
ComboBox	Show		
	Hide		
	Index	selection #	
	Count		
	Get		
Dock	Float		

Widget Type	**Command**	**Parameters**	**Result**
	Dock		
	Move	x position, y position	
	Resize	width, height	
List	Count		
	Index	select posn 1, select posn 2, select posn 3, select posn 4	
	Select	select posn 1, select posn 2, select posn 3, select posn 4	
	Get	row #	
Label	Get		
LineEdit	Get		
	Clear		
	Set	text	
	Append	text	
SpinBox	Get		
	Set	value	
Splitter	GetSize	position	
	SetSize	position, size	
	Count		
	Orientation		
Table	Get	row, column	
	Clear	row, column	
	Set	row, column, text, more text	
Tabs	Count		
	Get		
	Set	tab #	
TextEdit	Clear		
	Get		
	Append	text	

Widget Type	**Command**	**Parameters**	**Result**
View	Mouse		
	Move	xpos, ypos	
	Click	xpos, ypos	
Window			
MessageBox	Detail		
	Info		
	Text		
	Title		
FileDialog	Select	filename	
	SetDir	directory name	
FontDialog			
ColorDialog			

8 Interface Testing

The aim is for each sub-project to be able to do its testing in isolation. Since the modules interact the other modules need to be simulated for us to do an isolated test.

8.1 Installing Test Modules

The simulated modules are installed using the registerFactory() method of the FactoryFactory class. They take over the slot for the existing factory so that whenever a request is made for the module the simulated version will be returned.

```
FactoryFactory &ff = FactoryFactory::instance();
ff.registerFactory(Editor_Module,
             Ed::ViciEditorFactoryPtr(new ViciEditorIfTestFactory) );
```

8.2 Simulating Events

The simulated modules can register their functions with a Dispatcher object. They are indexed by the use case and the name of an event. This allows the test case to initiate an event without needing to know exactly how the simulated module implements it.

To register a function:

```
// register events
stub::Dispatcher &dispatcher = stub::Dispatcher::instance();

dispatcher.registerEvent("uc102", "invalidate",
      new stub::AnEvent<ViciAdminEbnfTest>(
                   &ViciAdminEbnfTest::uc102a, this));

dispatcher.registerEvent("uc102", "validate",
      new stub::AnEvent<ViciAdminEbnfTest>(
                   &ViciAdminEbnfTest::uc102, this));
```

To trigger an event from a test case:

```
stub::Dispatcher::instance().sendEvent("uc102", "invalidate");
```

Of course the test case will still need to poke around to verify that objects are in the expected state.

8.3 Event Tracing

The Dispatcher has methods for collecting trace information. You can turn on tracing, trigger an event, and then compare the sequence of function calls to what was expected.

This currently works, but probably conflicts with the cfd::Trace component. It does have the advantage that the protocol can be verified without having the test case know much about the simulated modules.

Update this so that it only records intermodule events.

9 Testing Installed Programs

The aim is to verify that the installed programs are working properly.

We will use the plug-in test mechanism to run tests. The programs will be started (perhaps by a VICI script) which will set the configuration file to a test version (vici-test.xml). This will have references to installed plug-in libraries and scripts for the GUI Test Harness.

The Test Lab will use this mechanism for doing bulk testing across distributions and desktops.

More to go here once we have the process working.

10 Logging

The cfi library provides a variety of logging mechanisms. These can be accessed as follows:

```
logstream &log = logstream::instance(logName);
```

and then log can be used just like any other stream.

The logName in the above example is used to access the configuration file to determine how and where the data will be written, for example the Tester uses the following:

```
<LOG>
      <TestLog>
            <type>unformatted</type>
            <file>@abs_top_builddir@/scratch/logs/test.log</file>
      </TestLog>
</LOG>
```

Please refer to the Programmer's Guide section on Logging Classes for the available alternatives.

10.1 Network Logging

One of the options for logstream is for a UDP link to a logging server. This server is vici-logger. The configuration file provide the server name and port number for vici-logger.

The Test Lab will probably use this to collect the logs from the virtual machines.

Need to see if Tester can use upd logging.

11 Tracing

A significant problem in maintaining a C++ program is understanding how the various objects interact. The aim of the tracing component is to provide collaboration diagrams that show these interactions.

11.1 Trace Macro

The core of the tracing component is a macro which placed at the start of each method. This macro creates a local object using the name of the function as a parameter. The constructor and destructor make calls to the Tracer object which logs these events.

This macro needs to be modified so as to use the __PRETTY_FUNCTION__ macro so that it can be inserted without any editing.

11.2 Trace Logging

There can be a lot of output from tracing and it can impact performance. It will be enabled by a configure variable available in config.h.

A new means is needed to restrict the functions being traced. The current scheme using the configuration file is not practical.

11.3 Trace to Dot

The trc2dot program converts the trace logs into a graph for use by the GraphViz dot program. This creates an SVG file which is then processed by an XSL script to add animation.

Appendix A